NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

UTILIZING IXP1200 HARDWARE AND SOFTWARE FOR PACKET FILTERING

by

Jeffery L. Lindholm

December 2004

Thesis Advisor:	Su Wen
Co-Advisor:	John Gibson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) Utilizing IXP1200 Hardware and Software for Packet Filtering			**5. FUNDING NUMBERS**
6. AUTHOR(S) Jeffery L. Lindholm			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

As network processors have advanced in speed and efficiency they have become more and more complex in both hardware and software configurations. Intel's IXP1200 is one of these new network processors that has been given to different universities worldwide to conduct research on.

The goal of this thesis is to take the first step in starting that research by providing a stable system that can provide a reliable platform for further research. This thesis introduces the fundamental hardware of Intel's IXP1200 and what it takes to install both hardware and software using both Windows 2000 and Linux 7.2 as the operating system in support for the IXP1200.

This thesis will provide information on the installation of hardware and software configuration for the IXP1200 including Intel's Software Development Kit (SDK). Upon completion this platform can then be used to conduct further research in the development of the IXP1200 network processor. It provides a hardware and software installation checklist and documentations of problems encountered and recommendations for their resolution. Along with providing an example of using preexisting code that has been modified to filter packets of TCP or UDP to different ports.

14. SUBJECT TERMS IXP1200 ACE MicroACE, Intel IXA, Microengine			**15. NUMBER OF PAGES** 79
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

UTILIZING IXP1200 HARDWARE AND SOFTWARE FOR PACKET FILTERING

Jeffery L. Lindholm
Lieutenant, United States Navy
B.S. Computer Science, Hawaii Pacific University, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2004

Author: Jeffery L. Lindholm

Approved by: Su Wen
Thesis Advisor

John Gibson
Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As network processors have advanced in speed and efficiency they have also become more complex in both hardware and software configurations. Intel's IXP1200 is one of these new network processors that has been given to different universities worldwide to conduct research with.

The objective of this thesis is to take the first step in beginning the research by providing a stable system that can provide a reliable platform for further research. This thesis introduces the fundamental hardware of Intel's IXP1200 and the requirements for installation of both hardware and software using Windows 2000 and Linux 7.2 as the operating system in support for the IXP1200.

This thesis will provide information on the installation of hardware and software configuration for the IXP1200 including Intel's Software Development Kit (SDK). Upon completion, this platform will be able to conduct further research in the development of the IXP1200 network processor. It provides a hardware and software installation checklist along with documentations of problems encountered and recommendations for their resolution. Included is an example using preexisting code that has been modified to filter packets of TCP or UDP to different ports.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

During the course of my thesis work, there were many people who were instrumental in helping me. Without their guidance, help and patience, I would have never been able to accomplish this thesis. I would like to take this opportunity to acknowledge some of them.

I would like to thank my thesis advisors, Professor Win Su and Professor John Gibson, both of whom were driving factors in the successful completion of this thesis. Professor Win Su introduced me to Intel's IXP1200 that sparked my interest into the possibility of the card. Professor John Gibson was instrumental as an instructor in many of my classes that led to the completion of this thesis.

I must give immense thanks to my wife, Regina and our children Stefanie, Trenton, and Andrew. Their love and support during long nights of work away at the lab was of immeasurable value to me.

THIS PAGE INTENTIONALLY LEFT BLANK

I. BACKGROUND

A. THE EVOLUTION OF NETWORK PROCESSORS

Up until the late 1990's, most network routers were based on architecture similar to a personal computer. The central processing unit (CPU) performed the basic network layer processing tasks: access control filtering, forwarding table lookups, and routing updates. The central processor received instructions from the router's operating system that ran in random access memory (RAM), along with basic instructions that were stored in read only memory (ROM). The advantage to this architecture was that all instructions were stored in software; therefore, new features could be added simply by upgrading the systems software, much like in a PC. Routers could also be designed with additional interfaces that supported High Speed Serial Interface (HSSI), without having to touch the basic processor architecture. For vendors this meant that their basic router architecture could support a variety of configurations as well as different versions of the same basic operating system. Therefore, vendors were able to quickly develop new products with a reduced time-to-market and make routine changes at a low cost to support a small segment of the market. [1, 2]

A perfect example of this approach is the Cisco 2500 product line. This software-based router uses a central processor unit to perform routing instructions based on software configurations stored in nonvolatile RAM. The Cisco 2500 series is available in a variety of specialized hardware configurations depending on specific application requirements, such as, multiple or single serial ports, multiple or single attachment unit interface (AUI) ports, or multiple or single Token Ring ports. On the software side, dozens of different software versions are available to support highly specific individual requirements. [2]

The drawback to software-based routers is their limited ability of scale to support the demands of higher bandwidth and additional features. For example, most software-based routers currently available are only capable of supporting wire-speed throughput of less than a single 155 Mbps, or in some extreme cases, up to a single 622 Mbps. When these same routers are then tasked to perform complex traffic filtering, policy based

routing, or collection of traffic statistics, their performance suffers and their maximum throughput is greatly reduces. [6]

The use of Application Specific Integrated Circuits (ASIC) was a major paradigm shift for hardware manufacturers in the production of network devices. Rather than using software-based processing and improving performance by increasing the speed of the central processor, hardware manufacturers discovered they could achieve tremendous performance improvements by creating specialized chips. These ASIC chips are manufactured with embedded instructions, and therefore, perform forwarding decisions directly in hardware, which improves the overall performance. [11]

While ASIC-based switching has allowed for a new generation of very high-speed routers and switches, there is a downside to this approach. Once instructions or logic has been embedded into silicon, it is difficult to change them to add new features or to improve performance. This means that manufacturers must replace the ASIC chips to enable new functionality, unlike traditional software-based routers and switches in which new features can be added by simply upgrading the operating system. Another problem is that errors in the ASIC design during product development can result in substantial time-to-market delays since it sometimes takes a silicon factory a month or more to produce a new set of ASIC chips. While advanced features, such as complex quality of service routing, identification of upper layer flows, gathering of accounting information, or access control filtering, still requires traffic to be processed in software the performance benefits of the ASIC-based architecture is reduced. [12]

The demand for performance has given rise to the hybrid configuration, such as, network processors. The Intel's IXP1200 Network Processor family is an example of the new approach. The IXP1200 hardware is made up of one core processor the StrongARM and six RISC processors or Microengines that it manages. Since both the StrongARM and the Microengines are software driven and programmable this makes the development process faster. The combination of specialized machine instructions for traffic processing and its' ability to use parallel processing with the Microengines makes it possible to offer high performance along with versatility for change.

The IXP1200 contains a StrongARM core and six independent 32-bit RISC data engines (Microengines), as well as SRAM, SDRAM, PCI and IX bus controllers. The

operating frequency for the PCI bus is up to 166 MHz, where as, the IX bus operates between 166 and 232 MHz. The performance of the processor is advertised to be 3 million packets per second (3 Mpps), which is 1.5 Gbps (3000000 * 64 bytes * 8 bits). The combination of six ASIC-based Microengines, one programmable StrongARM processor, both SRAM and SDRAM, and a dedicated IX bus interconnecting all of its components has produced a single network-processing chip that is both programmable and very efficient. [11]

Synchronous Random Access Memory (SRAM) is a type of RAM that holds its data without external refresh for as long as power is supplied to the circuit. This is contrasted to dynamic RAM (DRAM), which must be refreshed numerous times per second in order to hold its data contents. SRAMs are used for specific applications within the PC, where their strengths outweigh their weaknesses compared to DRAM. SDRAM incorporates new features that allow it to keep pace with bus speeds as high as 100 MHz. It does this primarily by allowing two sets of memory addresses to be opened simultaneously. Data can then be retrieved alternately from each set, eliminating the delays that normally occur when one bank of addresses must be shut down and another prepared for reading during each request. [19]

Network Processors are designed to take advantage of the inherent parallelism in the packet processing programs. Most of them provide additional/replicated resources on the chip in order for packets to be processed in parallel. For example, the Intel IXP1200 network processor has six Microengines on the chip. These are simple RISC processors that run independently and can accelerate functions like hashing, calculations, lookups, and so on.

Network processors have been used as an attractive alternative to pure PC-based routers. Even though their architectures can be used to support network monitoring, a newer much more performance-oriented architecture is needed.

B. THE INTEL IXP1200 NETWORK PROCESSOR

This section will briefly introduce the architecture of the IXP1200, derived from the Intel IXP1200 Network Processor Family Hardware Reference Manual. An exhaustive hardware specification can be found at Intel’s website,

http://www.intel.com/design/network/products/npfamily/index.htm. Figure 1 is provided as a reference while discussing IXP1200 configuration.

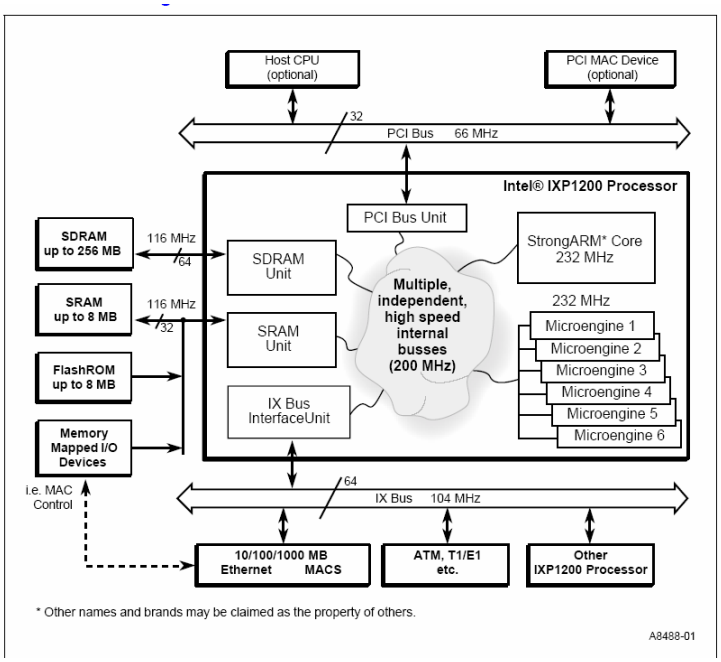

Figure 1. Intel IXP1200 Diagram (From Ref 11)

1. Memory

The IXP1200 has three types of memory resources available to the programmer: SRAM, SDRAM, and Scratchpad RAM. These memories vary in latency, size, and bandwidth. SDRAM memory is typically used for storage of packet data and very large tables; whereas, SRAM memory is used for table lookups where low latency is very important. Scratchpad RAM, being internal to the IXP1200, is smaller but with very low latency and is most often used for inter-process communication, and shared semaphores or counters to enable Microengines to share resources. [11]

a. SDRAM

The SDRAM unit is for interfacing to external SDRAM via a 64bit x 116Mhz bus. It shares an intelligent memory interface that can be accessed by the StrongARM, Microengines, and other devices on the PCI bus. It is a flexible interface to standard, low-cost SDRAMs that are capable of moving blocks of data between the SDRAM and the Microengines by using the IX Bus Unit, or devices on the PCI bus. [3].

b. SRAM

The SRAM unit interfaces to external SRAM via a 32bit x 116Mhz bus. It shares an intelligent memory interface accessible by the StrongARM and the Microengines. It provides a flexible interface to standard pipeline and flow-through. SRAMs are capable of moving blocks of data between the SRAM and the Microengines by using the IX Bus. [4].

2. StrongARM Core Processor

The core processor for the IXP1200 is a 32-bit StrongARM RISC processor. It may be used in two different ways depending on the application and host CPU that is available. In configurations where there is no host processor or the host processor is limited, the StrongARM would act as the main processor running a Real Time Operating System, performing system maintenance functions, as well as, running as the core processor. Alternately, the StrongARM can leave the system maintenance processing to the host processor of the CPU, acting as only the exception processor, performing higher-layer processes for the Microengines. The latter is the configuration that is used for this paper. The StrongARM runs a mini-kernel and executes routing protocols, while the Microengines do the fast-path packet processing. [11]

The operating system functions of the StrongARM boot from Flash memory over the SRAM interface. The StrongARM can then load the Microengines with their own programs. The StrongARM communicates with the Microengines via shared registers and a shared on-chip Scratchpad RAM. It also has the ability to enable or disable any of the Microengines or any of the Microengine threads. The StrongARM can make changes to the Microengine instruction store at any time by disabling the Microengine, and writing to their instruction store and then restarting that Microengine. The granularity of the StrongARM's ability to manage the Microengine reaches down to the individual threads running on the Microengines. [11]

3. Microengines

The six Microengines operating at 232Mhz are compact and efficient RISC engines that are used for any function requiring high-speed packet inspection, data manipulation, and data transfer. These 32-bit engines, with a 5-stage execution pipeline and a large (256) register set, are fully programmable. The Microengine is designed in

such a way to provide multi-threaded operations enabling each Microengine the ability to run up to four threads at the same time. The Microengine runs on a lower language of assemble code. It is within the Microengines that the speed of processing is achieved. The StrongARM is much like the manager in a grocery store with the cashiers being the Microengines. The cashiers handle the majority of the daily tasks in the most efficient way possible. It is the StrongARM or the manager that handles complex or unexpected tasks and leaves the Microengines free to take care of other tasks. [11].

4. IX Bus

The IX Bus is the main interface for receiving and transmitting data with external devices, such as, MAC devices and other IXP1200s. It is 64 bits wide and runs up to 104MHz allowing for a maximum throughput of 6.6Gbps. The Microengines can directly interact with the IX bus through an IX Bus Unit; so a thread running on a Microengine may receive or transmit data on any port without StrongARM intervention. This interaction is performed via Transmit and Receive FIFO queues, which are circular buffers that allow data transfers directly to/from SDRAM. For the Microengines to interact with peripherals, they need to query or write to control status registers.

5. Fast Bus Interface (FBI)

The FBI unit contains several parts that are 1K of memory of 32 bit words on the scratchpad, a hash unit and the interface to the IX bus which connects the network interface card to the IXP1200. The memory is divided into two parts the Receive memory and the Transmit memory. Both receive and Transmit memory act as a buff between the host computer and the IXP1200. [15]

6. Host Computer

The IXP1200 is contained on a PCI card that is mounted into a host computer. The IXP1200 card requires that the host computer use Linux 7.2 as its operating system to enable the IXP1200 card and the host computer to communicate with each other. This communication is done through the PCI bus which the IXP1200 card is attached too. When the IXP1200 card is in place, the host computer provides power to run the card, hard disk space for storage of programs or operating instructions. The PCI interface provides communication between the host processor and the IXP1200. With the

IXP1200 card installed it operates as an extinction of the host computer so all communication to and from the IXP1200 is done through the host computer. [11]

C. SUMMARY

Network processors started as a central processing unit that used RAM and ROM to store its operating system and routing tables to maintain connections to the networks. This very simple system has evolved along with the networks that we have today to a very ridged system of removing RAM and ROM to designing the programming into hardware during the manufacturing process of the network processors. Along with using multiple processors that are programmable to accomplish a single task in parallel while enabling both speed of processing and programming flexibility.

Intel developed the IXP1200 network processor board that consists of multiple programmable processors. It is made up of two buses, the PCI bus and the IX bus. The PCI bus is used for communications between the host computer and the IXP1200 card. While the IX bus is the bus system on the card providing the communication link between the components on the card. The card is made up of one master processor, the StrongARM, which supports six processors the Microengines that run independently. All are programmable being able to provide both increased network processing speed and the programmability for change.

The next chapter will cover the functionality of the IXP1200 software and its installation.

THIS PAGE INTENTIONALLY LEFT BLANK

II. INTEL'S SOFTWARE DEVELOPMENT KIT AND INSTALLATION

Intel's Software Development Kit (SDK), Version 2.0, is made up of a C compiler Cygwin, which is a command line interface; and VMware, which is a virtual machine that runs on Windows to support the Linux operating system. More information on VMware can be found at http://www.vmware.com/products/ or information on Cygwin can be found at http://cygwin.com/. Within the SDK, Intel provided a GUI called The Developer's Workbench that provides help with programming and with the operation of the IXP1200. These tools are combined to provide complete resources needed for programming, including a resource of libraries for code reuse, modification, and as a development environment to assist the programmer.

The Developer's Workbench is a key component of the SDK that features a GUI based simulator for the Microengines and hardware of the IXP1200 network processor. The simulator is used as a test bed for the Microengine code as it is developed. Within the Workbench there are many pre-developed Active Computing Elements (ACEs) provided. ACEs are similar to a macro or fragments of code that have already been developed and tested for functionality and efficiency. These ACEs may be combined to produce code to be run on the IXP1200. ACEs that are written in C or C++ can be run directly on the StrongARM. The ACEs must be in the appropriate assembler language in order to run on the Microengines. This is the main benefit of the Workbench, in that it is the only available tool to accomplish the task of compiling code for the Microengines. The process to produce code for the Microengines is to take ACEs that are already written from the Development Workbench and combine them into a single functioning piece of code that then is compiled using Workbench into a list file that will be run on the targeted Microengine.

The Workbench is capable of debugging code using the target hardware or simulating a packet flow for testing code. The debugger can be used to debug the code to be run on the Microengines. Within the debug mode there is the option of setting breakpoints, single step commands, and jumping over a function. Other features of the Developer's Workbench are access to a command line window, performance statistics,

facilities to save and restore simulation session, memory modify history, data watch window, and thread history window.

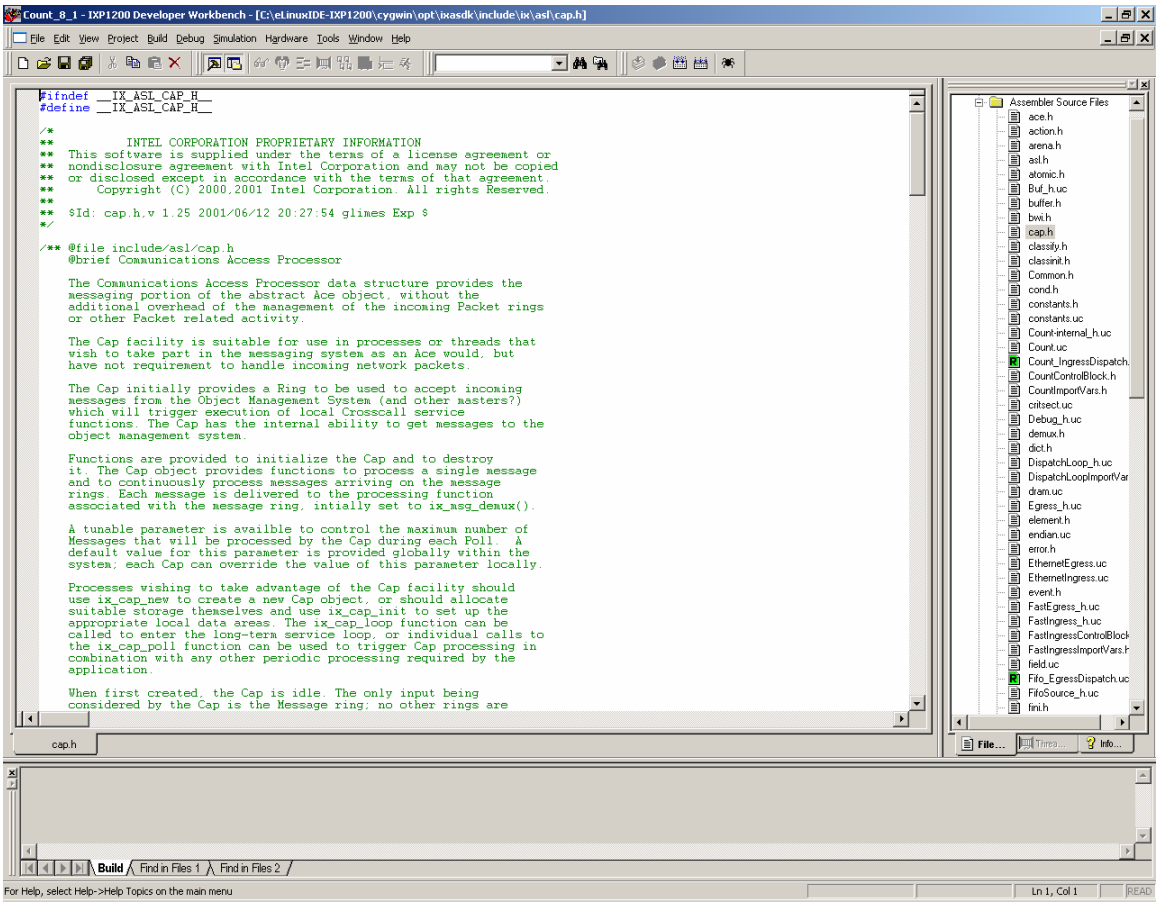

Figure 2. IXP1200 Workbench GUI (Ref 18)

Figure 2 shows a typical window of the Developer's GUI Workbench. It looks and acts much like Microsoft's Visual Studio, but there are many differences because the target hardware configuration includes one core processor and six supporting processors. Below are some of the more important parts of the functions of Workbench that will be described in greater detail

A. BUILD SETTINGS

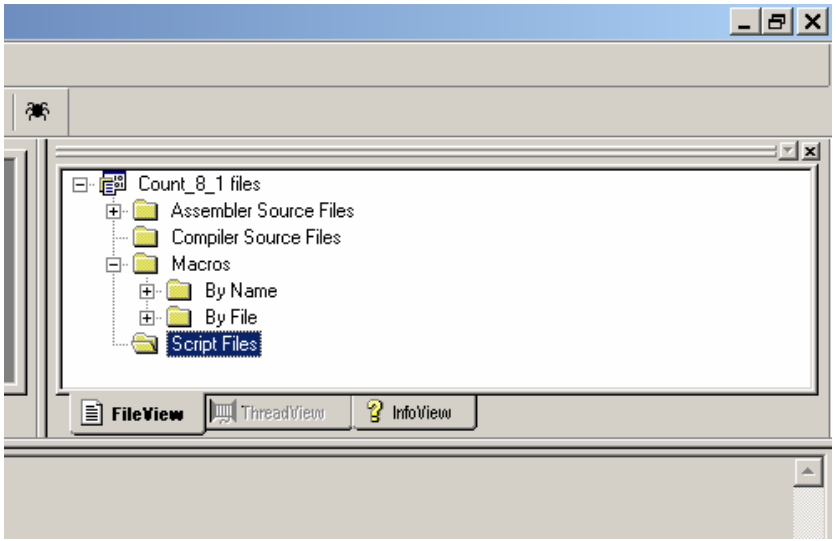

Figure 3. Project Workspace

Figure 3 shows the file manager in Workbench. The source files for both the assembler and the compiler are listed alphabetically in their own file. Source files can be added by clicking on the project bottom or by providing an include statement similar to C++. Macro are listed in two files by the name of the macro and by the name for easy access. One of the benefits of Workbench is that it provides this file of commonly used

macros that can be added to the project with minimal effort.

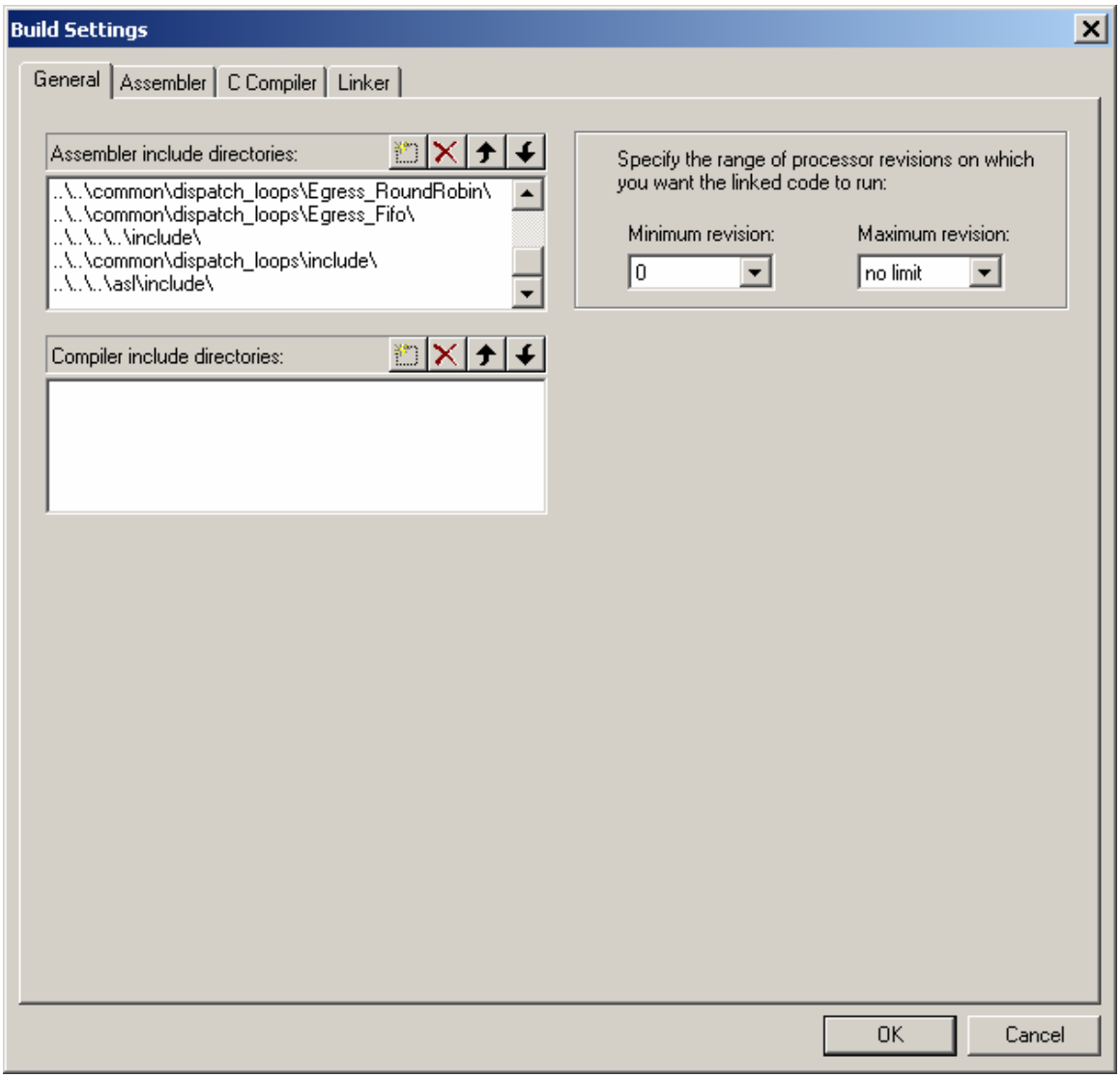

Figure 4. General Build Settings

After assembly code is written, additional settings must be configured to build and run the program. In Figure 4 there is an example of the window from the development workbench for the general build settings. It is here that the compiler is told were to look for the files that are needed to build and compile the code for the project.

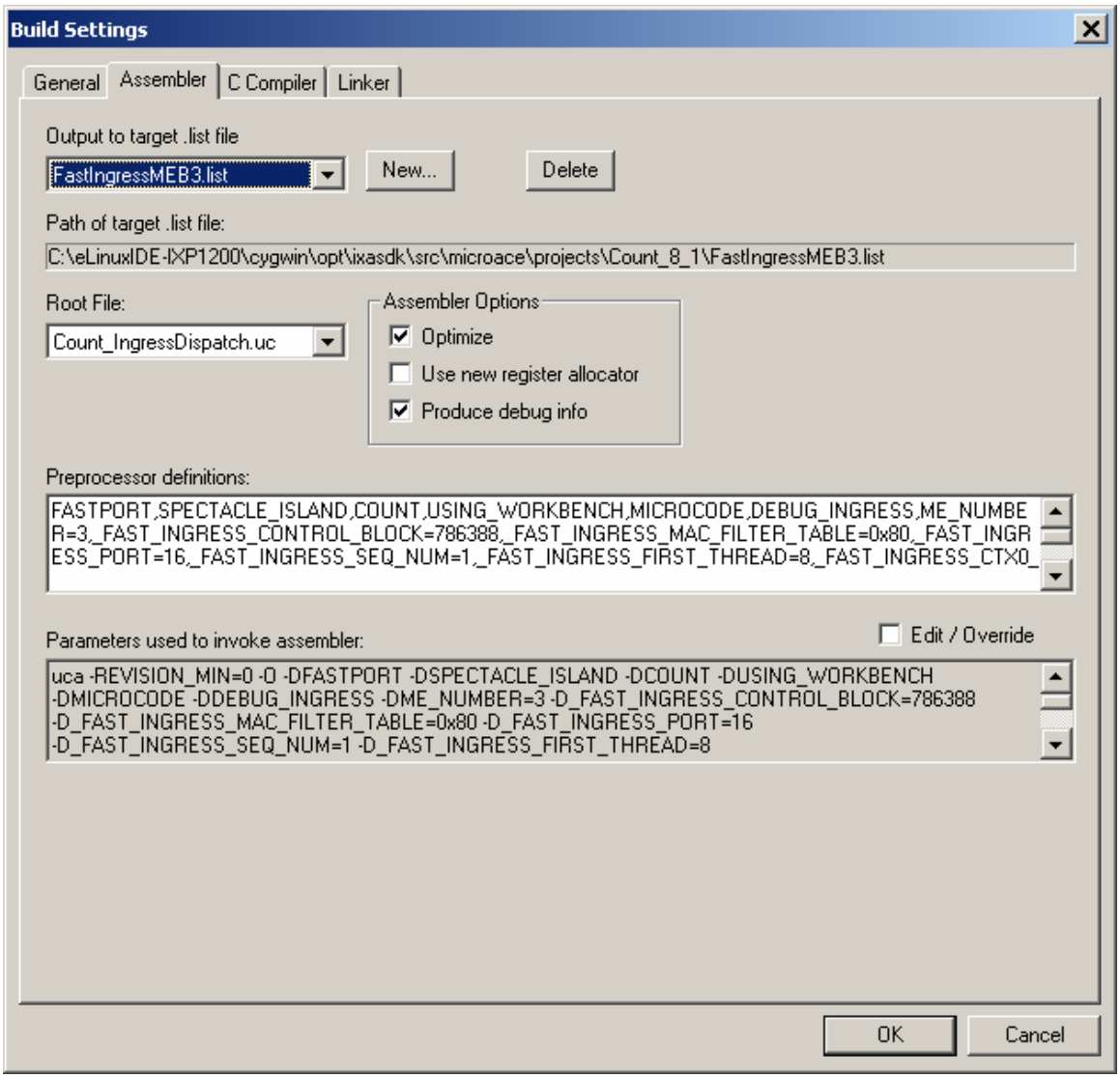

Figure 5. Build Settings Assembler

Within the window displayed in the Figure 5 are the options to assign the root file and the target list file. The Root file is the much like the main in C++ that the project will run from many times it is a dispatch loop. The target list file is the place that the compiler will send the compiled code to be run on the Microengines.

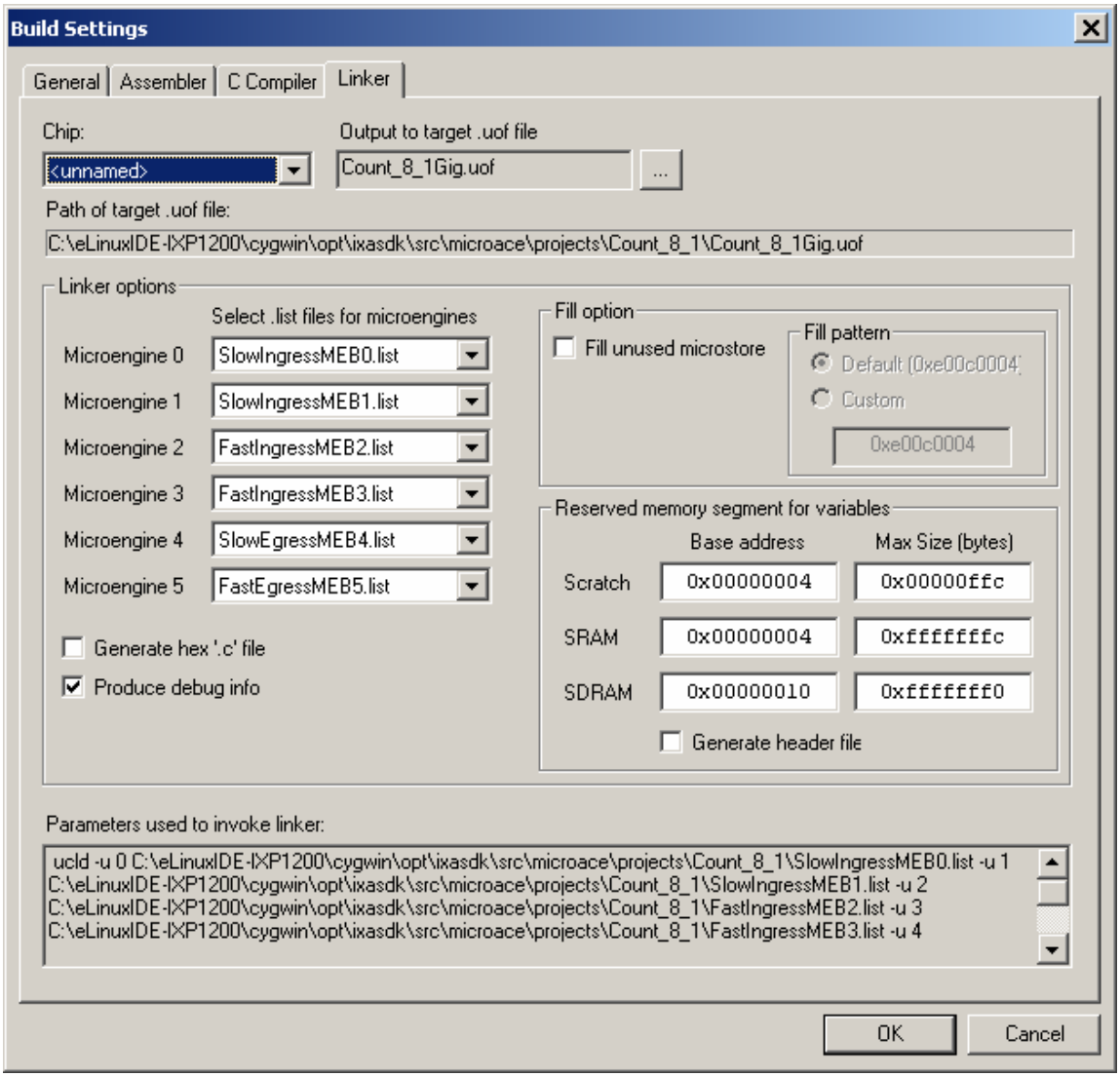

Figure 6. Build Setting Linker

Figure 6 contains the decision as to which Microengine will run what list file. The list file is is shown in the Linker tab (Figure 6). This is how to tells the compiler, which Microengines will run what list file for the program. It is important to consider while assigning the list files to the Microengines that you manage resources like memory and Microengines to reduce bottle neck in the project. The simulator is a very good tool found within the debugger to find and reduce bottlenecks.

B. DEBUGGING CODE

The debugging section of the Development Workbench is a very useful tool. The debug section provides tool for tracking history of the program as it runs. This history can be tracked all the way down to each thread that is run at the same time. Other functions of the debug are command line inter face and a data watch to track variables.

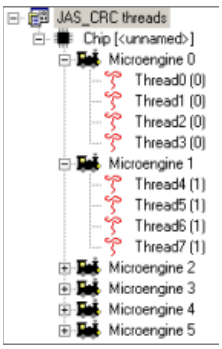

Figure 7. Thread View

Figure 7 is an example of the thread view in the debug mode.

The Thread View window provides a graphic view of the six Microengines and the four threads that can be run on each Microengine.

Info View provides access to documentation on the Workbench components, such as, the Development Tools User's Guide and the Programmer's Reference Manual.

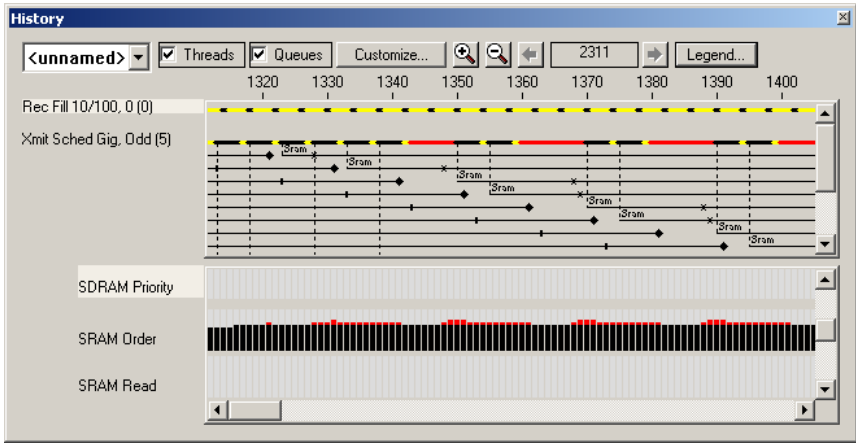

Figure 8. Thread and Queue History Window

Figure 8 is the Tread and Queue History window that provides a view of the Microengines executing code, making it possible to quickly locate performance bottlenecks. The threads and queues are displayed on a timeline that represents the number of cycles executed. The line segments on the timeline change color depending on whether an instruction is executing, stalled (waiting on resources to complete a task), aborted, or idle (waiting for a task to be assigned).

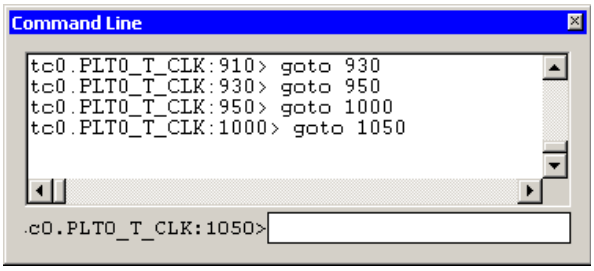

Figure 9. Command Line

Figure 9 shows the command line interface. It has two functions, one for simulation mode and the other for hardware mode. In simulation mode the command is passed to the simulator, which in turn sends the command and the response back to the command line window. In hardware mode the command line provides an alternate way to access the hardware. Some of the commands that it supports are C interpreted functions, script files and conditional directives.

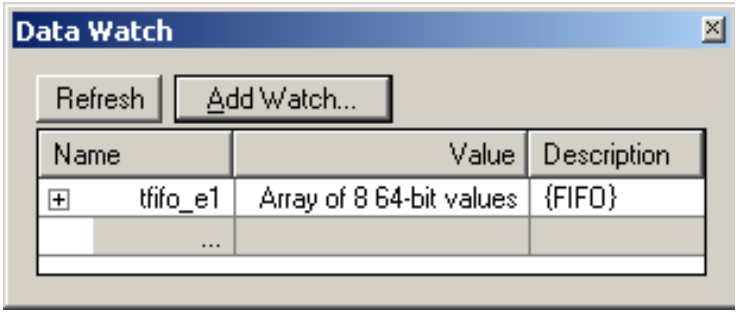

Figure 10. Data Watch

Data Watch is useful in tracking the value of variables (Figure 10). To view a particular variable, enter the name of the variable in the Name block of the data watch window. Data Watch will update the variable when the Microengine or thread execution has stopped.

The network simulator can be used to run new code to compare run times. One of the nice things about the simulator is that it will produce line graphs of different simulations for comparison. This should prove very useful in a multi-thread environment when it is necessary to find code bottlenecks. It also has a very extensive library of tested code that can support code reuse. The debug section of the SDK can separate each thread of the program, both graphically and as it steps through the code.

C. INSTALLATION OF IXP1200 CARD AND SOFTWARE

The host machine requires a modified version of Linux 7.2 operating system in order to run and communicate with the StrongARM. The StrongARM uses a very limited version of the Linux operating system. The resources available to the IXP1200 are very limited. For this reason, the StrongARM uses the host computer's hard drive to store its files. The Linux C compiler is used to compile code to be run on the StrongARM. Linux, or Cygwin, if a version of Windows operating system is running on the host, is used to move files from the host machine's directory to where the StrongARM can read them.

For the installation of the IXP1200 card and software needed to support the card refer to Appendix B. Appendix B provides a complete checklist for installing the SDK Windows operating system and Linux 7.2 with Cygwin.

D. SUMMARY

Intel's Software Development Kit (SDK) contains a GUI known as the Developer's Workbench. This Developer's Workbench is the key component of SDK which features a GUI based simulator for the Microengines and the hardware for the IXP1200 network processor. Its main function is to test the programs it develops for the Microengines within which are many pre-developed Active Computing Elements (ACEs). These already proven and tested bits of codes or macros are combined to produce code to run on the IXP1200. In order for the Ace to run on the Microengine the Workbench modifies the ACE to an assembler level.

The Workbench is able to debug code using the target hardware and simulating packet flow for the code being tested. Within this mode there are several options, such as, setting breakpoints, single step commands, and jumping over functions. Other important features of the Developer's Workbench, such as, command line window,

performance statistics, save and restore simulation, memory modification, data watch window, and thread history window. This development tool is set up in such a way to take a complicated situation and making it simpler with the reuse of code.

III. PROGRAMMING PACKET FILTER ON IXP1200

Programming the IXP1200 is substantially different then a typical sequential line program that has only one processor with which to deal with when writing code. The IXP1200 is designed to be programmable and multi–threaded, thus able to run tasks in parallel Additionally, it can run independent from the host computer's processing unit. Being programmable allows great flexibility to change or modify operation through programming. Multi-threading provides the possibility of great processing speed. To program in this kind of an environment is very complex due to potential memory conflicts, since more then one processor uses the same memory. Getting the different processors to work together to complete tasks efficiently is also a challenging task. Intel has reduced the complexity of coding in this environment with the use of Active Computing Elements (ACEs).

ACEs are small chucks of code that have been perfected to be run on the IXP1200. They can be used as building blocks which are assembled to complete more complex operations. These ACEs have already been tested and perfected for efficiency and reliability. This makes it possible to assemble ACEs with minimal effort and resulting in a reliable and efficient product following the paradigm of code-reuse.

There are two categories of ACEs: ACEs and MicroACEs. ACEs are run on the StrongARM, and solely written in C and C++, while MicroACEs are simply an ACE that has been compiled to run only on the IXP1200 Microengines. The MicroACEs and Microengines form the fastest processing path. As noted above, they're assembled together or linked together to complete tasks, in this way they work as a team. Each one has its own function to complete then calls the next ACE which calls the next ACE until the whole process is complete. Below is more detail concerning the process of creating code and using ACEs.

A. DESIGNING A PACKET FILTER APPLICATION

In today's Internet the speed of handling traffic is paramount to being efficient because of the increased amount of traffic that is present on it. In order to filter packets to or from the Internet all packets have to go through the filter. If the filtering process is slower than the routing process, the filter will become a bottleneck. The IXP1200 has

two main components on which code is run, the StrongARM and the Microengines. Both have to be considered when producing code. Both support the use of ACEs in the process of developing code for the IXP1200.

ACEs are the building blocks that make up the code that runs on the IXP1200 card. In terms of functions there are three types of ACEs running on the StrongARM: User ACEs, Library ACEs, and System ACEs. User ACEs provide the user interface to the system. Library ACEs are those provided by the Workbench or Intel to produce common functions for use within the code. System ACEs implement network interfaces, protocol, and algorithms. Control ACEs, a subcategory of System ACEs, produce routing tables.

When dealing with packets, an ACE does three things: it classifies the packet, acts on the packet or disposes of the packet. An ACE also communicates with other ACEs through queries. These queries allow each ACE to monitor the status of such things as traffic, completion of tasks, and memory management.

ACEs encourage modularity, reducing the complexity of code verification and validation, because an application can be implemented by chaining several existing ACEs together to achieve the desired functionality. If the ACEs are already debugged and tested, a programmer can be confident that by using these ACEs the code that is produced will be both efficient and need limited debugging. Such modularity is possible due to the Library ACEs, which by using them reduces the duration of the coding process. Another benefit is that ACEs can be reused in many applications without having to be recreated from scratch. An example can be seen in the packet filter in Appendix A which was created with Library ACEs with little difficulty.

When designing an application to run on the IXP1200 it is useful to divide it up so it will be manageable. One suggestion is to design the different functions and ACEs along with establishing the packet flow for the application. Producing an integrated application involves the following general steps.

First, define the functions and packet flows of the ACEs that run on the StrongARM and MicroACEs that run on the Microengines. This must be accomplished for each Microengine used by the application. Next, the programmer must determine which ACEs will be executed by the Microengines and which will be executed by the

StrongARM, running as the core processor. At this point a dispatch loop is written to implement the packet flow within the application. Microblocks are simply a group of ACEs put together to complete a task or function.

Dispatch loop configures the group of functions in a Microblock into a processing pipeline. The dispatch loop along with code in a Microblock is compiled into a single ".uof" file that will be downloaded to the Microengines to be run as an application. It then decides how many Microengines to load with each image of the code. This depends on the port configuration of the application. Each Microengine is able to run four threads and each 10/100 Mbps port requires one thread. Thus, each Microengine can run four standard or fast Ethernet ports. However, a gigabit port requires eight threads, which means two Microengines are required. All of this is done in the Workbench that can then provide the means to test the program for efficiency. Care must be taken to ensure that overhead operations are limited to reduce bottlenecks caused by the sharing of resources. For this reason it is best to limit the use of dividing Microblocks among different Microengines. [18]

B. MPACKETS AND MEDIA ACCESS CONTROLLER

The IXP1200 network processor card must break down all input packets into 64-byte Mpackets in order to be used by the IX bus, which is the primary data path for the IXP1200.

The Media Access Controller (MAC) segments the original packets into Mpackets and reassembles them upon completion of the filter processing. The first MPacket will have the label, Start of Packet (SOP), prep ended while the last will have the label, End of Packet (EOP), appended. An original packet of length 64 bytes or less will have both labels affixed and be seen by the Microengines as a whole packet. Original packets longer than 128 bytes will be fragmented such that the first and last MPackets generated will have the appropriate label affixed, but interior MPackets are unlabeled.

The MAC puts the Mpacket on the IX bus; from there it goes to the Fast Bus interface (FBI) to feed the Microengine. The Microengine then must reassemble the original packet from the Mpackets using a buffer in the SDRAM. The buffer holds sequential Mpackets until an Mpacket is received with the label EOP at which time the restored packet is complete and can be processed.

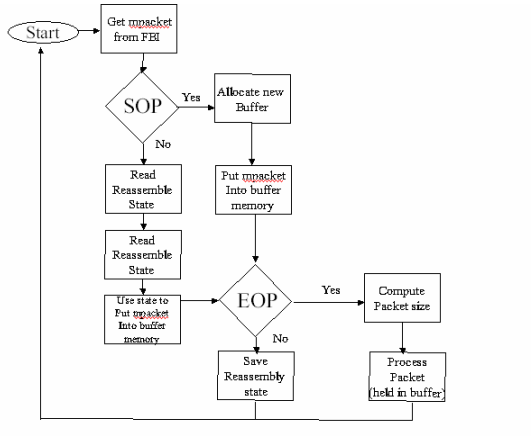

Figure 11. Flow chart for Mpackets (From Ref 1)

C. PROCESSING PACKETS

The processing of packets can include several Microblocks and, depending on how they are assigned to parts of the IXP1200 network processor, can have a dramatic effect on system performance. When assigning Microblocks to Microengines, a single Microblock should be processed on a single Microengine, due to the added cost of inter-process communication if the Microblock crosses Microengines. Further, up to four Microblocks can be processed simultaneously on a single Microengine. More than four Microblocks on a single Microengine results in extra overhead, as the various blocks must be scheduled to run according to the limitation of four concurrent threads that are possible to be run on each Microengine. Running packets of different sizes on the same Microengine also increases overhead cost as the Microengine has to assemble the Mpacket packets from the IX bus. It is best to assign differing packet types to separate Microengines since they must be processed differently. This helps to limit the overhead involved in the process. [5]

1. Dispatch Loops

Dispatch loops are used to separate different types of packets and control the flow of packets. Below is pseudo code representing a dispatch loop.

```
While(True){
        Get next packet from input device(s);
Invoke ingress Microblock;
If (return code ==False){
        Drop the packet;
} else if (return code == True) {
        Send packet to ingress core component;
} else{/*IP packet*/
        Invoke IP Microblock;
        If(return code ==False){
                Drop packet;
        } else if (return code ==True) {
                Send packet to IP core component;
        } else {
                Send packet to egress Microblock;
        }
}
```

This dispatch loop is an endless loop that gets the packet from the input device (IX bus). It starts a Microblock that checks the packet to see which of three options is appropriate: 1) drop the packet, 2) send it to a Microblock to be checked again, or 3) send it to another Microblock to be processed. The second choice of the loop calls an IP Microblock and then further divides the processing, based on the result of the IP Microblock, into three more additional choices: 1) drop the packet, 2) send it to a core component, or 3) send it to an egress Microblock to be processed.

The ingress source block gets a packet from a port and records the input port in a global register using a dispatch loop macro, and then passes the packet to the Layer 3 Forwarder Block.

The Layer 3 Forwarder Block checks whether the packet is an IP packet, an ARP packet, or another type of packet. For every IP packet, the Layer 3 Forwarder will validate the packet. If the packet is invalid, it will be dropped. If the packet is valid but

has options in the header, is fragmented, or is multi-cast it will be sent to the StrongARM ACE component of the Layer 3 Forwarder. This communication mechanism is handled via the dispatch loop and the resource manager. Otherwise, the next-hop MAC address and output port number will be obtained from the forwarding table. The output port number is recorded in a global register using a dispatch loop macro. [14]

If the packet is not sent to the StrongARM, then the dispatch loop will queue the packet for egress. There is one queue per port and the dispatch loop will look at the output port to determine which queue to use. The egress Microblock removes the next packet from the queue and sends it out over the appropriate port. [14]

If a packet is sent to the StrongARM, the L3 StrongARM ACE component processes the packet and sends it to the egress output interface ACE using the bound target. The egress ACE then sends it down to the egress Microblock using a resource manager call. Alternatively, the L3 ACE may choose to send the packet back down to the L3 Microblock using the resource manager based on the criteria set.

D. FILTERING TCP AND UDP

Appendix A contains the packet filter for TCP and UDP packets that will be discussed below. This packet filtering application has three dispatch loops: Global Registers, Macros, and Application. The Global Registers dispatch loop supports each Microblock that sets and accesses the dl_buffer-handle and dl_next_block global registers. The Macros dispatch loop provides predefined macros that can be used to perform the following actions: access cached variables, assign input and output ports for the buffered packet, dispatch a buffered packet by dropping the packet or sending the packet to a different Microblock group, or follow the transfer of packets between the Microblock and the core component of a MicroACE. The Application dispatch loop defines constants and declares registers to be used within the dispatch loop. TCP/UDP filtering can be implemented by modifying a function with in the Count program, an example program provided by the IXP1200 SDK. Appendix A shows the modified function that can be used to implement the TCP and UDP packet. The original Count function can be found in the Intel Internet Exchange Architecture Software Development Kit CD 1. The path for this project that was modified is /opt/ixsadsk/src/microace/project/Count_8_1_2. The only modifications that were made

to the project count_8_1_2 was to the Count.uc file, which defines the processing function being call in the Count_IngressDispatch.uc, the ingress dispatch loop file of the project.

Below is a line by line narrative of the modified code for Count.uc, as provided in Appendix A.

1. The application Count defines constants that are to be used in the program.
2. It declares both local and global registers. Global registers are accessible to other Microblocks enabling inter-process communications. After this is completed a call is made to include the other functions.
3. Initializes each Microblock in the Microblock group.
4. Starts the packet filter function (the Filter() macro is a modification of the original Counter() macro).
5. Checks for packets passed down from the core MicroACEs. This is done by polling the core processor to queue the Microengine.
6. Checks if a packet was retrieved from the queue.
7. Returns the data pointer for the packet's buffer and handles returns: The data pointer (into SDRAM).
8. Addition of the pointer to the packet data in SDRAM.
9. Directs the pointer to the correct location to obtain the necessary type of packet.
10. Allocates four continuous registers and treats them as a one single string.
11. Takes the numeric value that is in the field and places it into the variable type.
12. Checks to see if it is an IP packet or another type of packet.
13. Branches for IP packet and other types of packets
    a. IP packet
        i. Allocates eight new registers in SDRAM to store the packet while working on the IP header
        ii. Reads the IP header
        iii. Assigns a variable to store the numerical value of the protocol type in.

iv. Reads the value of protocol type and stores it.

v. The variable is read and the loop branches depending on the Value.

1. Value 6 indicates TCP and is sent out port two
2. Value 17 indicates UDP and is sent out port four
3. Any other value is sent out port one as an exception

b. Ethernet sent to port one.

E. COMPILING AND RUNNING CODE

1. Compiling the Microcode Source Files

The first step is to compile the Microcode source files using the Malefile.win. This can be achieved using Cygwin on the windows operating system. Cygwin is a command line interface that is used on the windows machine to manipulate the files in the Linux format.

From the command line the command echo is used to ensure the $IXROOT and $CONFIG are correct, such as, "echo $IXROOT and echo $CONFIG". $IXROOT should be /opt/ixasdk and $CONFIG should be ARM_BE. If not, the command export is used to correct them to "export IXROOT=/opt/ixasdk or export CONFIG=ARM_BE".

Next a change in directories is needed to $IXROOT/src/microace/aces/tutorial1/ucbuild. This is done with the command "cd $IXROOT/src/microace/aces/tutorial1/ucbuild".

Once the directory is correct compile the Microcode to make the .uof files. The command to compile these sources is "make –f Makefile.win".

Now getting these .uof file to where the IXP1200 card can read and use them is necessary. It may be possible to use FTP, but I did not have any luck doing that and used a copy command on the command line in Cygwin, such as, "scp **.uof** **root@192.168.0.4:/opt/ixasdk/bin/arm-be***"

2. Compiling Core Component of MicroACE

Another change of directories is needed to $IXROOT/src/microace/aces/tutorial1/count_ace1 using the command "cd $IXROOT/src/microace/aces/tutorial1/count_ace1". Now ensure $IXROOT is

/opt/ixasdk, $config is ARM_BE and $IXPSDKROOT is c:/ixp1200. Complete just like above.

Next in the command line type the command “make” then change directory to $IXROOT/src/microace/projects/Count_8_1

3. Running the Application

Reboot the host computer in Linux. Then Open a command window on the host computer the IXP1200 card is remotely accessed from this command window by typing “cd /opt/ixasdk/enp-2505/bootixp”, next Type “./bootixp” to boot the IXP1200. To check the status of the IXP1200 to see if it is up and running use the command “ifup” from the same command window that was used above that should be now the command window for the IXP1200. The command “ifup” not only relays the IP address, but starts the port mapper and the debugger. The StrongARM of the IXP1200 needs to know were to access its files from the host computer this maybe done by using the terminal window of the IXP1200 card. The mount command is used to achieve this by configure the StrongARM to retrieve its code by entering the command, “mount –tnfs <host IP address>:/opt/ixasdk/bin/arm-be mnt”. Then change to directory mnt. At this point the application is ready for use. To start the new project the command is “./ixstart <your config file>.

The configuration file is what the StrongARM uses, while run from the ixstart. The configuration file has all the parameters that are needed to run the project. These parameters consist of were and what .uof file are to be loaded onto which Microengine for the project. With in the parameters or configurations can be changed things like the number of ports and IP address that are available to the IXP1200 card. Theoretically a programmer can make simple changes to the congregation file to modify the project by assigning new .uof file that are already compiled.

F. SUMMARY

Programming the IXP1200 involves many components. The Windows environment supports the Intel's Development Workbench and allows the management and debugging of Microcode. The Linux environment allows the programming of core processor the StrongARM, while providing an interface with hardware. There are many variables involved to succeed: hardware must be connected correctly so compiled c and Microcode executables can be loaded, and the configuration file correctly set for the program to finally run on the IXP1200 card. To run the IXP1200 effectively the programmer has to consider the use of multiple threads running at the same time in order to manage memory and deadlock that can be created in the environment. This chapter presented the steps and modifications needed to implement a TCP/UDP filtering function on the IXP1200.

IV. CONCLUSIONS AND RECOMMENDATION

Network processors started as central processing units using basic RAM and ROM to store its operating system and routing tables to maintain connections to the networks. Over time network processors have evolved, such as, Intel's newly developed IXP1200 network processor card that consists of multiple programmable processors. It contains two buses for communications, the PCI bus talks between the host computer and IXP1200 card while the IX bus is the link between components on the card. The card has one master processor, the StrongARM, which supports six processors the Microengines that run independently. All are programmable and are able run parallel processing which has the ability to increase network processing speed and be programmable for change.

The Developer's Workbench is the key component of Intel's Software Development Kit which features a GUI based simulator for the Microengines. Its purpose is to test and developed programs for the Microengines which contains pre-developed, proven code known as Active Computing Elements (ACEs). Because of ACEs the IXP1200 is able to run tasks in parallel. This is one way it differs from the typical sequential line program

The IXP1200 card is a very complex piece of equipment that requires precise, detailed instructions to install; as well as the process of creating and changing preexisting programs to run on the IXP1200 card. Tools, such as, the Developer's Workbench provided by Intel makes it possible to take preexisting code and with simple modifications produce good running code for a specific task by using code reuse. The IXP1200's ability to reuse code worked quite well when modifying Intel's count program into a packet filter for TCP and UDP.

There were problems that arose because of the complexity of the IXP1200 system. Below is a list of problems that were discovered and recommendations.

In today's world of security and numerous upgrades to operating systems, difficulties arose between the two different operating systems in their ability to talk. Intel's answer to making the systems talk was FTP. In my research, I was unable to achieve this function in the development process and had to come up with other options like copying files from one IP address to another IP address.

The IXP1200 card 9 pin serial connector uses COM1 for communications to the host PC, therefore it can cause conflicts with Windows mouse. This conflict arose with Windows 2000 which checks for new hardware at startup and periodically thereafter. During this check, Windows assumes that the connection on comport one is a serial connection for a mouse. The IXP1200 card uses COM1 as its COM port and if the connection is changed to COM2 or another serial connection the card will not boot correctly.

After performing updates for Windows 2000, Cygwin encountered errors when starting the program and would not continue running. To correct this problem required that all of the SDK be removed before reinstalling Cygwin. Due to the fact Windows sees Cygwin as part of the SDK; therefore, not allowing it to be removed and reinstalled by itself.

One should not recompile the grub-1-0.src.rpm provided by the Intel installation disk. There is an error in the file that destroys the boot file for Windows 2000, which makes it impossible to boot Windows. If this file was recompiled, when starting Windows the following message would appear: "Windows 2000 could not start because the following file is missing or corrupt: <windows 2000 root)\system32\ntoskrnl.exe. Please re-install a copy of the above file". For this reason VMware cannot be installed. Further research revealed that possibly using Windows NT version 4 vice Windows 2000 may fix this problem. Note: There are many versions and configurations of IXP1200 and SDK out on the web which adds to the confusion of finding the answer to fix this problem.

While installing Windows 2000 on top of Linux there's a possibility of losing the grub file, but there is an uncomplicated fix for this. In Linux command window type "grub", enter "root(hd0,1)", then enter "setup". This will set the grub file back to the default.

When the kernel for Linux 7.2 is recompiled to support the IXP1200 it limits the function of the original kernel. Problems were encountered with mounting the A drive or floppy. To solve this two versions of Linux were ran, the original 7.2 and the upgraded 7.2 version for IXP1200.

There are immense possibilities for IXP1200 in the future with the ability of parallel processing and being able to program in a timely manner with the use of code reuse within Intel's SDK. The problem is making people aware of the IXP1200 and developing a foundation of knowledge to perfect new beneficial uses for it.

My recommendation is to first create a stable platform of computers or computer lab for the IXP1200 and then provide a class for programming the IXP1200. This class would expose more students to IXP1200 thus, creating more interest in it and subsequently, increasing the knowledge base for the IXP1200 card.

By creating this lab and giving the students a base at which to start from would expedite the learning process and any further research to be done with the IXP1200. Only with increased exposure will the true potential for the IXP1200 card be reached. The IXP1200 possess the ability to be programmable. Along with its six Microengines' ability to each run four continuous threads in a parallel processing nature, the possibilities are unlimited with the versatility and the speed it can provide.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. TCP AND UDP PACKET FILTER

```
#ifndef _Filt_
#define _Filt_
#include "stdmac.uc"
#include "field.uc"
#include "xbuf.uc"
#include "endian.uc"
#include "ip.uc"
#include "Buf_h.uc"
#include "CountControlBlock.h"
#include "Count-internal_h.uc"

#include "Debug_h.uc"


//8888888888888888888888888888888888888888888888888888888888888888
// #include "FiltImportVars.h" is the header file that has the variables for filter.
//8888888888888888888888888888888888888888888888888888888888888888
#include "FiltImportVars.h"
#macro Filter_Init()
#endm
//88888888888888888888888888888888888888888888888888888888888888888
//  Filter is the name of the macro being called from the dispatch loop
//88888888888888888888888888888888888888888888888888888888888888888
#macro Filter()

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//         .local assigns local variables to be used in this dispatch Loop.
```

```
//888888888888888888888888888888888888888888888888888888888888
.local input_port exception_code stats_addr data_ptr data_offset type
expected_type

//888888888888888888888888888888888888888888888888888888888888
// Description:
//      This macro is used to obtain the port on which the packet being processed
// was received.  Valid range is 0 to 255.
//888888888888888888888888888888888888888888888888888888888888
        DL_GetInputPort[input_port]

//888888888888888888888888888888888888888888888888888888888888
// Description:
//      Assigns a place in memory for stats addr to track the number of packets
//888888888888888888888888888888888888888888888888888888888888

        immed32[stats_addr, _Filt_STATS_BLOCK]

//888888888888888888888888888888888888888888888888888888888888
// Description:
//      Increments stats_addr for each packet that is taking in from the input port
//888888888888888888888888888888888888888888888888888888888888

        Filt_IncrementPacketFilter[stats_addr]

//888888888888888888888888888888888888888888888888888888888888
// Description:
//      This macro returns the data pointer for the given buffer handle returns:
//The data pointer (into sdram)
//888888888888888888888888888888888888888888888888888888888888
         Buf_GetData(data_ptr, dl_buffer_handle)
```

```
//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      This macro is used for the purpose of obtaining the offset of the packet
data from the starting point of the packet buffer.  This is combined with
Buf_GetData to obtain the pointer to the packet data in SDRAM.
//8888888888888888888888888888888888888888888888888888888888888888
        DL_GetBufferOffset(data_offset)

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      This macro is used by the alu to direct the pointer to the correct location to
// obtain the type of packet.
//8888888888888888888888888888888888888888888888888888888888888888
        alu[data_ptr, data_ptr, +, data_offset, >>3]
//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      ???
//8888888888888888888888888888888888888888888888888888888888888888
        Debug_Assert(data_offset & 0x7 == 0)

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Allocates four continuous registers and treats them as a one single string
//8888888888888888888888888888888888888888888888888888888888888888
        xbuf_alloc($$l2_header, 4)

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
        Instruction which reads the first two quad words from SDRAM starting at
data_ptr.  The destination is the continuous registers that are allocated.
```

The ctx_swap value is a flag to the Microengine to swap this thread out until this memory request is satisfied.

```
//8888888888888888888888888888888888888888888888888888888888888888
    sdram[read, $$l2_header[0], data_ptr, 0, 2], ctx_swap

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction that takes the numeric value that is in the field and places it into the variable type.
//8888888888888888888888888888888888888888888888888888888888888888
         xbuf_extract(type, $$l2_header, 12, 0, 2)

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction that places a variable with a constant value in the register expected_type.
//8888888888888888888888888888888888888888888888888888888888888888
    immed32(expected_type, ETH_TYPE_IP)

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction that frees up the register $$12_header
//8888888888888888888888888888888888888888888888888888888888888888
         xbuf_free[$$l2_header]

//8888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction that uses the ALU subtract type from expected_type answer places is placed in br!
//8888888888888888888888888888888888888888888888888888888888888888
    alu[--, type, -, expected_type]
```

```
//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction checks to see if it is an IP or Ethernet.  A 0 indicates a Ethernet
packet which makes the program jump to the label mepass.
//888888888888888888888888888888888888888888888888888888888888888
    br!=0[mepass#]

//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction allocates eight new registers in SDRAM ($$ip_header) to be
used to work on the IP header
//888888888888888888888888888888888888888888888888888888888888888
        xbuf_alloc($$ip_header, 8)

//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction reads the header of the IP packet into the register $$ip_header.
The sig_done value is a flag to the Microengine to swap this thread out until this
memory request is satisfied.
//888888888888888888888888888888888888888888888888888888888888888
        sdram[read, $$ip_header[0], data_ptr , 1, 4], sig_done

//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction wait for the read of the IP header.
//888888888888888888888888888888888888888888888888888888888888888
        ctx_arb[sdram]

//888888888888888888888888888888888888888888888888888888888888888
// Description:
```

```
//      Assigns a variable proto
//88888888888888888888888888888888888888888888888888888888888888888
        .local proto

//88888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction that takes the numeric value that is in the packet type field and places it into the variable proto.
//88888888888888888888888888888888888888888888888888888888888888888
        xbuf_extract(proto, $$ip_header, 15, 0, 1)

//88888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction assigns the value of 17 to UDP and 6 to TCP
//88888888888888888888888888888888888888888888888888888888888888888
        immed32(UDP,0x17)
        immed32(TCP,0x6)

//88888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Instruction checks to see if proto is TCP or UDP and then jumps to the appropriate place.
//88888888888888888888888888888888888888888888888888888888888888888
        if (proto == TCP)
                br[TCP#]

        elif (proto == UDP)
                        br[UDP#]
        else
                br[exception#]
#endif
```

```
        .endlocal

//888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Label for TCP packet
//888888888888888888888888888888888888888888888888888888888888888888
TCP#

//888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Sets the out going port to port two
//888888888888888888888888888888888888888888888888888888888888888888
DL_SetOutputPort[2]

//888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Resets port two to be used after the packet is sent.
//888888888888888888888888888888888888888888888888888888888888888888
immed[dl_next_block,1]

//888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      End of the jump or br
//888888888888888888888888888888888888888888888888888888888888888888
br[end#]

//888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Label for UDP packet
//888888888888888888888888888888888888888888888888888888888888888888
UDP#
```

```
//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Sets the out going port to port four.
//888888888888888888888888888888888888888888888888888888888888888
DL_SetOutputPort[4]

//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Resets the port four to be used after the packet is sent.
//888888888888888888888888888888888888888888888888888888888888888
immed[dl_next_block,1]

//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      End of the jump or br
//888888888888888888888888888888888888888888888888888888888888888
br[end#]

//888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Labels mepass for the first check if ethernet packets sent packets out port
one
//888888888888888888888888888888888888888888888888888888888888888
mepass#:
        immed[dl_next_block, 1]
        DL_SetQueueNum[input_port]
    br[end#]

//888888888888888888888888888888888888888888888888888888888888888
// Description:
```

```
//      Label for exception to handle everything that is not covered by the other
checks then sends in out port one
//88888888888888888888888888888888888888888888888888888888888888888888
exception#:
   immed[dl_next_block, IX_EXCEPTION]
        immed[exception_code, 5]
        DL_SetExceptionCode[exception_code]
        DL_SetQueueNum[input_port]

//88888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      Label for end that frees up $$ip_header and restarts the loop port one
//88888888888888888888888888888888888888888888888888888888888888888888
end#:

        xbuf_free[$$ip_header]
.endlocal

//88888888888888888888888888888888888888888888888888888888888888888888
// Description:
//      End of the macro
//88888888888888888888888888888888888888888888888888888888888888888888
#endm

#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. INSTALLATION OF HARDWARE AND SOFTWARE

A. SYSTEM REQUIREMENTS

This setup is for a dual boot system operating on Red Hat 7.2 and Windows 2000. This development environment is a combination of Windows 2000 and Linux platform or Cygwin for Linux environment on the Windows 2000 operating system that provides the following list of tools:

1. Microengine tool chain contains an embedded Linux version of the IXP1200 Microengine Development Environment, running on Windows 2000. The IXP1200 Card has to have the modified version of Linux 7.2 to run, but Microengine Development Environment allows the user to simulate code and debug before running the code on the IXP1200 card in the Linux operating system.
2. StrongARM tool chain is a GNU/CPP cross-hosted tool chain that can be called from a command shell or through a Embedded Linux IDE, which allows the user to develop and debug code running on the StrongARM on the IXP1200.
3. Omni NFS Sever makes directories located on Windows platform visible to application code running on the StrongARM.
4. Linux IDE supports application development, kernel debugging and device driver development.

The Linux side of the system needs Red Hat 7.2 to be modified for a host operating system for the IXP1200 board. Earlier and later versions of Red Hat will not support the IXP1200 card.

B. INSTALLATION OF WINDOWS 2000

1. Begin by formatting half of the hard drive space for the windows operating system then install Windows 2000.

2. From windows control Panel choose Users and Passwords

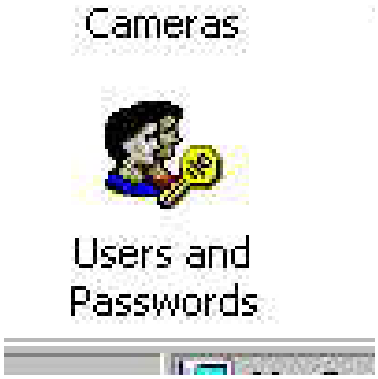

Figure 12. User and Passwords

2.1 Add the user ixa and remember the password

3. Log off to install Linux

C. INSTALLATION OF RED HAT 7.2 FROM CD ROM

1. Acquiring Red Hat 7.2

1.1. This can be down loaded from a web sight at URL http://six.retes.hu/download/redhat/. The files must be burned to a disk so that they can be used as a boot-disk during the installation process.

Note: The Windows operating system must be installed first in order to use the grub function.

2. Insert the number one disk into the CD drive.

Note: The boot directory in the BIOS may need to be reset. To get to the BIOS there will be an option as the computer boots up. (CTRL-ESC, F1,ALT-Tab)

3. Select the language of the user of the computer

3.1. Click Next

4. Select model of keyboard to be used

4.1. Click Next

5. Select model of mouse to be used

    5.1. Click Next

6. A welcome window will appear

    6.1. Click Next

7. Select Installation Type “Custom”

    7.1. Click Next

8. Choose “Manually Partition with Disk Druid” to partition the hard drive

    8.1. Click Next

9. Click on Free space

    9.1. Click on New

10. Click on Mount Point arrow and select “/boot”

11. Click on File system Type arrow and select “ext2”

12. Click on Size (MB) block and enter “50”

13. Select Fixed Size

    13.1. Click OK

    13.2. Note: Warning click “add anyway”

    13.3. Click on New

14. Click on File system Type arrow and select “swap”

15. Click on Size (MB) block and enter “512”

16. Select Fixed Size

    16.1. Click OK

17. Click on New

18. Click on Mount Point arrow and select “/”

19. Click on File system Type arrow and select “ext2”

20. Click on Size (MB) block and enter “4096”
21. Select Fixed Size
    21.1. Click OK
22. Click on New
23. Click on Mount Point arrow and select “/home”
24. Click on File system Type arrow and select “ext2”
25. Click on Size (MB) block and enter “50”
26. Select “Fill to Maximum allowable size”
    26.1. Click OK
    26.2. Click Next
27. Select use GRUB as the boot loader
28. Select install Boot Loader record on “/dev/had Master Boot Record (MBR)
29. Enter “Windows 2000” for Boot label box
    29.1. Note: This label is for the operating systems in the GRUB boot loader. There is also the option to pick a default operating system by clicking the box to the left of it at the bottom of the screen.
        29.1.1. Click Next
30. No password is needed on the GRUB
    30.1. Click Next
31. Do not configure the Network
    31.1. Click Next
32. Select No firewall
    32.1. Click Next
33. Select Language support
    33.1. Click Next

34. Select Time Zone

    34.1. Click Next

35. Enter root password twice

36. Add a user “ixa

    36.1. Click “Add”

        36.1.1. Enter User “ixa”

        36.1.2. Enter Password “ The same one from Windows “

        36.1.3. Enter Confirm Password

            36.1.3.1. Click “OK”

Note: Under Account Names your user should appear.

        36.1.4. Click Next

37. Authentication Configuration

    37.1. Click Next

38. Package Group Selection ensure the selection at least:

    38.1. NSF File Server

    38.2. Software Development

    38.3. Kernel Development

    38.4. Windows Compatibility / Interoperability

Note: Remember the size of the installation on the bottom of the screen

        38.4.1. Click Next

39. Select Video Configuration

    39.1. Click Next

40. This is the point at which the installation will begin.

    40.1. Click Next

41. Insert Disk two

    41.1. Click OK

42. Select box to make a boot disk

    42.1. Click Next

43. Select Monitor

    43.1. Click Next

44. Select Color Depth “High Color (16 Bit)”, Screen Resolution “1024x768”, “Graphical”

    44.1. Click Next

D. UPGRADING RED HAT LINUX KERNEL

1. Log into Linux as root
2. Insert CD “IXA EDU
3. Using a command window type “cd /mnt/cdrom/Host_Linux_upgrade”
4. Using a command window type “rpm –recompile linux-24-17src.rpm”
5. Reboot to check if system is intact
6. Edit the /boot/grub/grub.conf

Replace with sample below

```
# grub.conf generated by anaconda
# Note:  It is not required that grub be rerun after making changes to # this file
# NOTICE:  There is a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,1)
#          kernel /vmlinuz-version ro root=/dev/hda3
#          initrd /initrd-version.img
```

#boot=/dev/hda

default=0

timeout=10

splashimage=(hd0,1)/grub/splash.xpm.gz

title Red Hat Linux (2.4.7-10)

root (hd0,1)

kernel /vmlinuz-2.4.7-10 ro root=/dev/hda3

title Red Hat Linux (2.4.17)

root (hd0,1)

kernel /vmlinuz-2.4.17 ro root=/dev/hda3

E. INSTALL IXA SDK WORKBENCH FOR WINDOWS

1. Insert “Intel Internet Exchange Architecture Software Development Kit CD 1 Version 2.01 Volume II CD 1”
2. Run“cd:\workbench\setup”

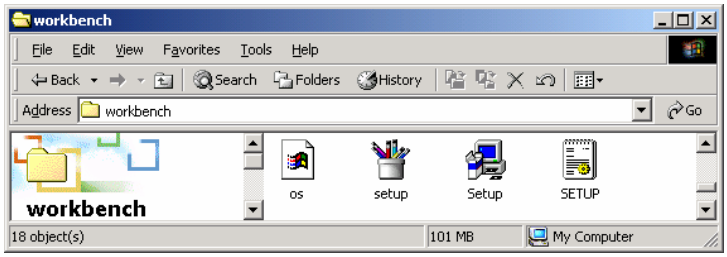

Figure 13. Setup Icon

3. There will appear an information label about IXP1200Portmapper and Windows NT

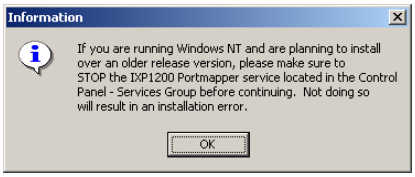

Figure 14. Information Window

3.1. Click OK

4. Use Alt-Tab to close all other program

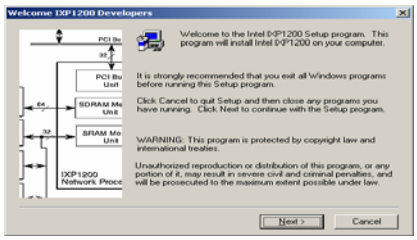

Figure 15. Information Window

4.1. Click Next

5. License Agreement

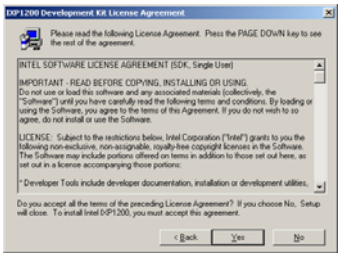

Figure 16. License Agreement

5.1. Click Yes

6. Setup location should be Drive “c:\IXP1200

Figure 17. Information on Path

6.1. Click Next

7. Select all components to install

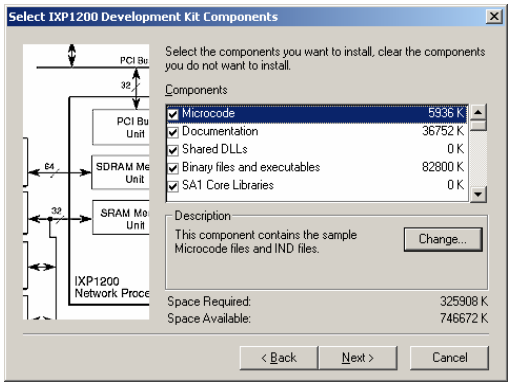

Figure 18. Select Development Kit Components

7.1. Click Next

8. Select program folder to place the startup icons in

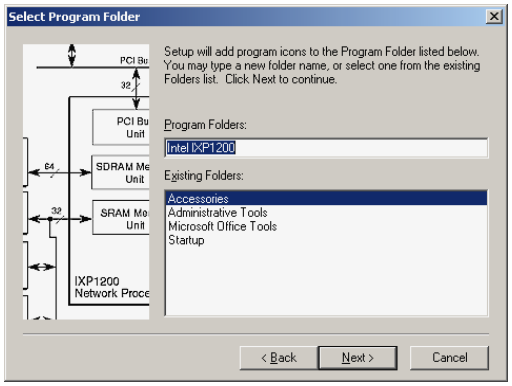

Figure 19. Select Program Folder

8.1. Click Next

9. Install IXP1200 PortMapper service

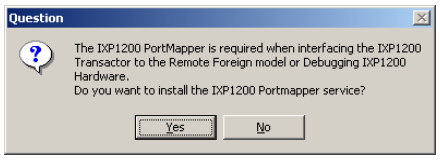

Figure 20. Question

9.1. Click Yes

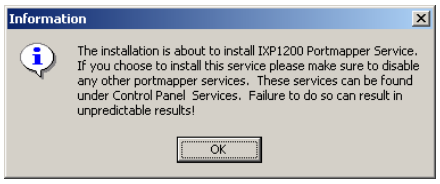

Figure 21. Information

9.2. Click OK

10. There will be a prompt for a key

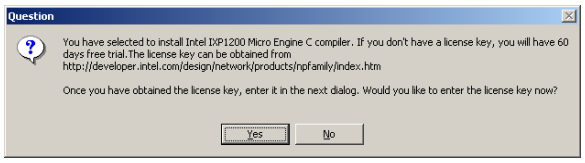

Figure 22. Question

10.1. Click Yes

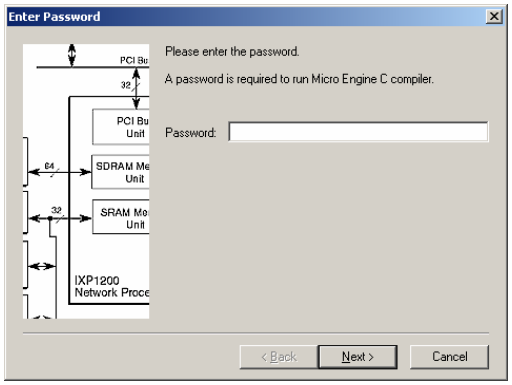

Figure 23. Enter Password

10.2. Click Next

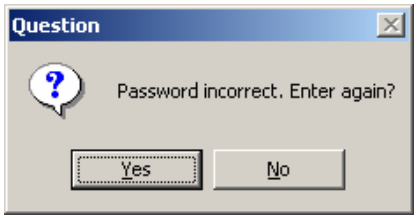

Figure 24. Question

10.3. Click No

11. Install Adobe Acrobat Reader

Figure 25. Information

11.1. Click OK

12. Workbench Installation Complete

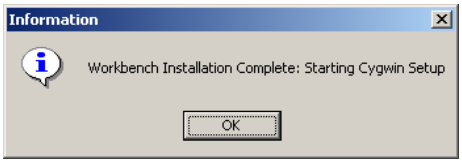

Figure 26. Information

12.1. Click OK

F. CYGWIN SETUP

1. Install Cygwin

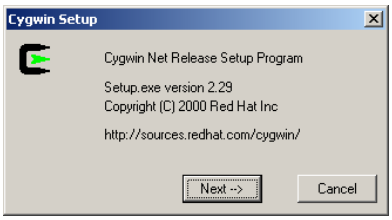

Figure 27. Cygwin Install

1.1. Click Next

2. Cygwin Install from Local Directory

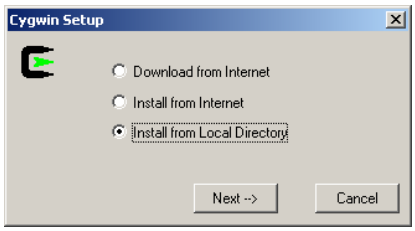

Figure 28. Setup Options

2.1. Click Next

3. Local Package Directory

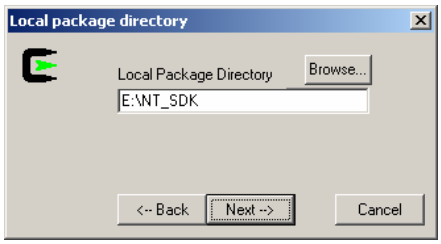

Figure 29. Directory Location of Cygwin

3.1. Click Next

4. Setup Configurations

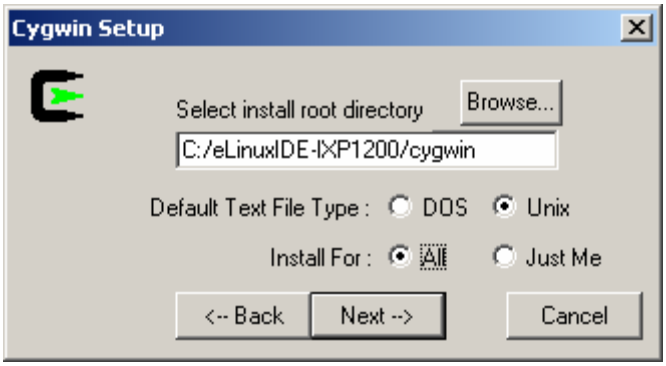

Figure 30. Setup User Options

4.1. Select Text File Type: Unix

4.2. Select Install for: all

4.3. Click Next

5. Select Packages to Install

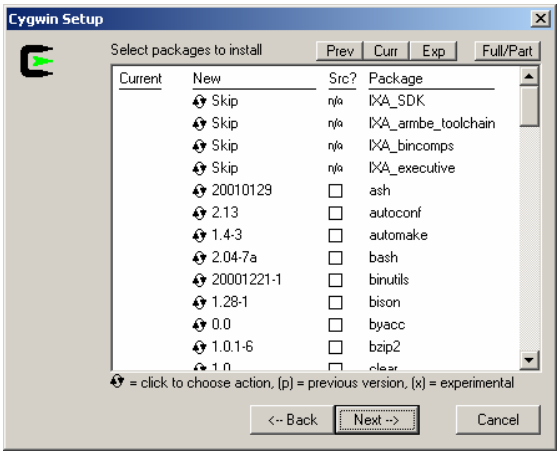

Figure 31. Component Options

5.1. Click fcs to skip the package

5.2. Skip IXA_SDK

5.3. Skip IXA_ambe_toolchain

5.4. Skip IXA_bincomps

5.5. Skip IXA_executive

5.6. Click Next

6. Short Cuts

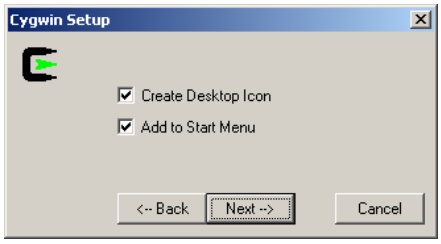

Figure 32. Icon Options

6.1. Click Next

7. Installation Complete

Figure 33. Installation Complete

G. INSTALLING OF IXP1200 CARD

1. Inspect the card for cracks or damage

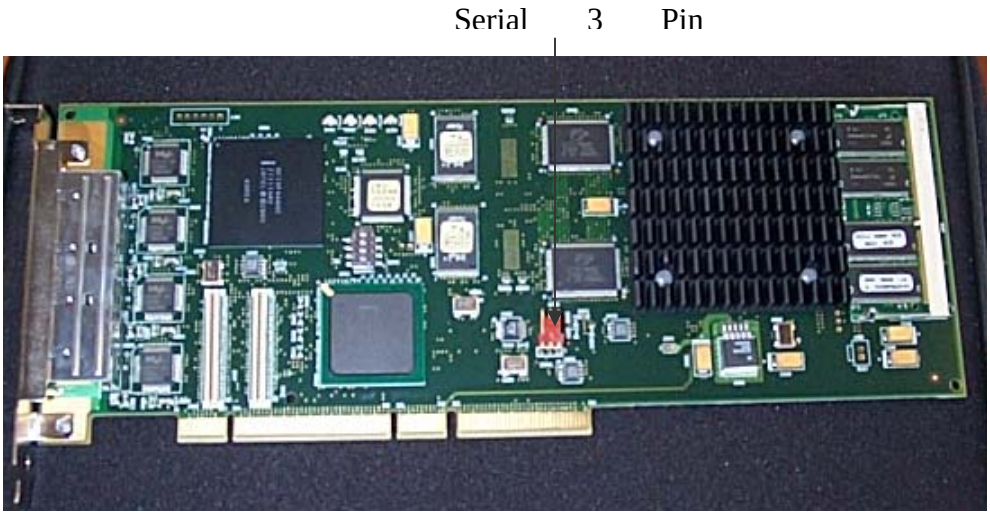

Figure 34. IXP1200 Card Inspection

2. Remove two expansion slot plate covers

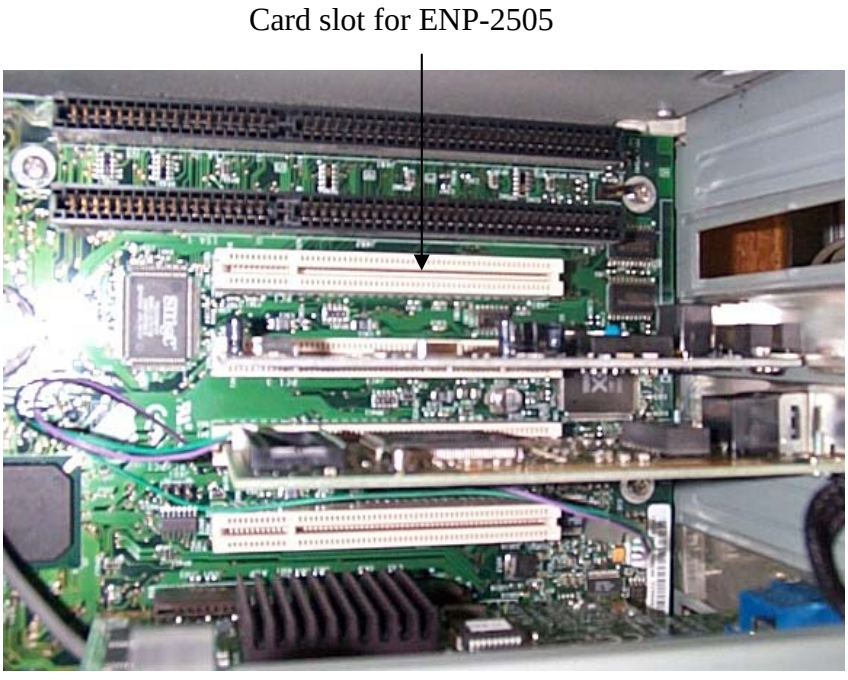

Figure 35. Mother Board

3. Install the three pin connector for the serial cable

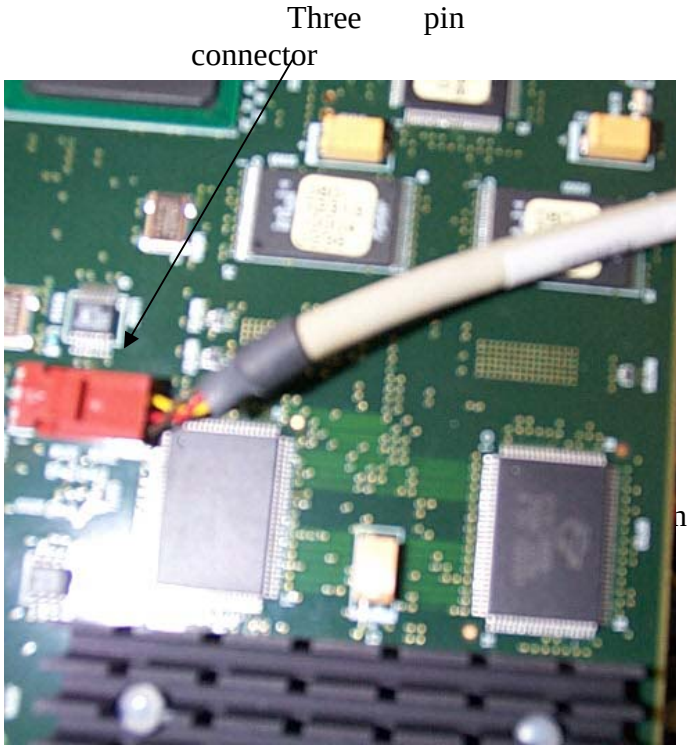

Figure 36. IXP1200 Three Pin Cable Connector

4. Install ENP-2505 card in the open slot

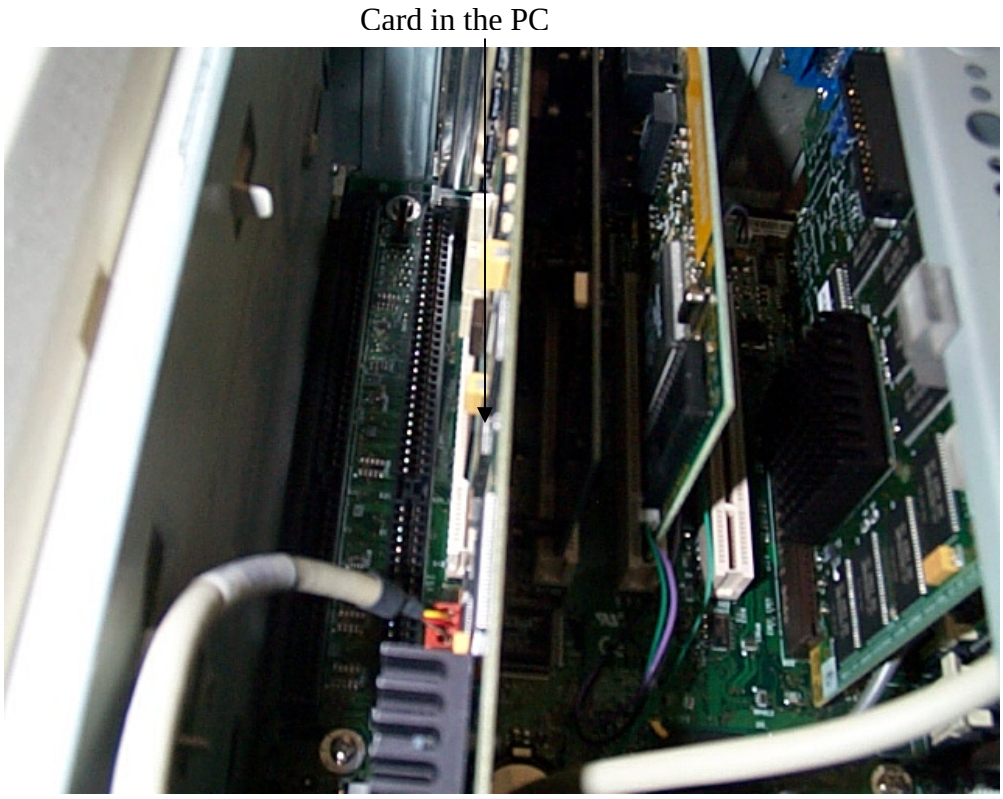

Figure 37. IXP1200 Card Installed

5. Install serial nine pin cable in the back of the PC

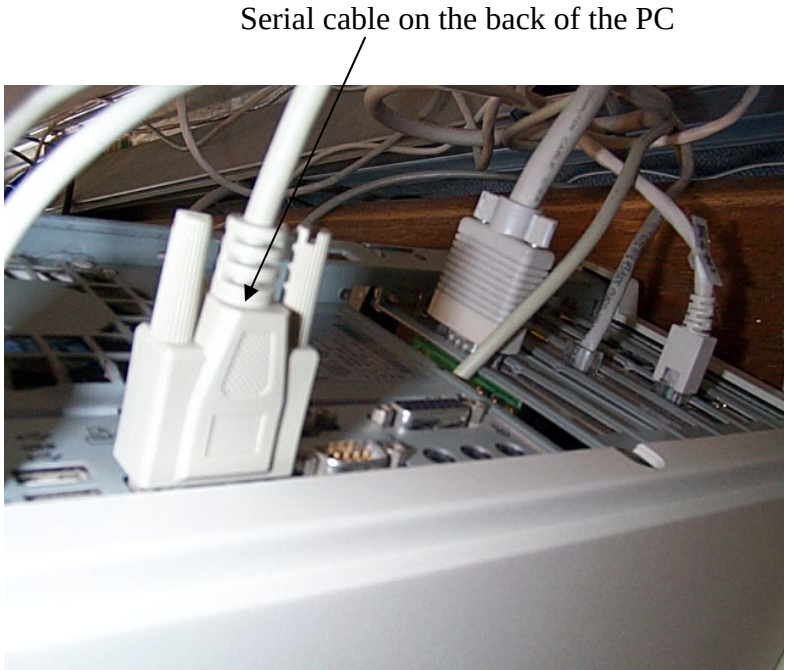

Figure 38. Connection of Serial Cable

H. INSTALLING LINUX STRONGARM DEVELOPMENT

1. Log into Linux as root
2. Insert CD disk 1 “Intel Internet Exchange Architecture Software Development Kit CD 1 Version 2.01 Volume II CD 1”
3. Open a command window
4. Type “cd /mnt/cdrom/rpms”

   4.1. Enter
5. Type “rpm –i ixa.sdk-2.01.D-fcs.i386/.rpm”

   5.1. Enter
6. Type “rpm –i ixa.executive-2.01.D-fcs.i386.rpm”

   6.1. Enter
7. Type “rpm –i armbe-v4b-fcs.i386.rpm”

   7.1. Enter
8. Type “rpm –i ixa.binaries-2.01.D-fcs.i386.rpm”

   8.1. Enter
9. Modify/ect/profile to make a path and environment variables permanent. To obtain access to the GNU tool chain, IXA-specific utilities and compilers the file “/etc/profile” must be modified
10. Open “/etc/profile”

    10.1. Replace /etc/profile with text below

```
# /etc/profile

# System wide environment and startup programs, for login setup

# Functions and aliases go in /etc/bashrc

# Path manipulation
```

```
if [ `id -u` = 0 ] && ! echo $PATH | /bin/grep -q "/sbin" ; then
    PATH=/sbin:$PATH
fi
if [ `id -u` = 0 ] && ! echo $PATH | /bin/grep -q "/usr/sbin" ; then
   PATH=/usr/sbin:$PATH
fi
if [ `id -u` = 0 ] && ! echo $PATH | /bin/grep -q "/usr/local/sbin" ;
then
    PATH=/usr/local/sbin:$PATH
fi
if ! echo $PATH | /bin/grep -q "/usr/X11R6/bin" ; then
    PATH="$PATH:/usr/X11R6/bin"
fi
# No core files by default
ulimit -S -c 0 > /dev/null 2>&1
USER=`id -un`
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"
HOSTNAME=`/bin/hostname`
HISTSIZE=1000
if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
    INPUTRC=/etc/inputrc
fi
# IXA SDK path setting
```

```
if ! echo $PATH | /bin/grep -q "/usr/local/armbe/bin" ; then
        PATH= "$PATH:/usr/local/armbe/bin"
fi
if ! echo $PATH | /bin/grep -q "/opt/ixasdk/bin" ; then
fi
IXROOT="/opt/ixasdk"
CONFIG="ARM_BE"
export USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC IXROOT CONFIG
for i in /etc/profile.d/*.sh ; do
   if [ -r $i ]; then
        . $i
   fi
done
unset I
```

I. ENP-2505 SOFTWARE INSTALLATION

1. Log onto Linux as root
2. Insert CD “IXA Education Workstation”
3. Open command window
   3.1. Type “cd /mnt/cdrom/ENP_2505_Driver
   3.2. Type “rpm –recompile boot_drv-2.0.-1.src.rpm
4. Reboot to Linux

J. BOOTING UP IXP1200 CARD

1. Open command window

1.1. Type “cd /opt/ixasdk/enp-2505/bootixp

1.2. Type “./bootixp

2. Check the Card with a ping command

2.1. Open another command window

2.2. Type “ping 192.168.0.4”

LIST OF REFERENCES

1. Erik J. Johnson, Aaron R. Kunze, “IXP1200 Programming”2002

2. Agere Systems Corp., “The Challenge For Next Generation Network Processors”, April 2001, http://www.agere.com/enterprise_metro_access/docs/challenge_new .pdf, last accessed 8/10/2004

3. Intel Corp.,”Internet Exchange Architecture Software Developers Kit 2.0 for the IXP1200 Network Processor”,2001, http://www.intel.com/design/network/prodbrf/27904101.pdf, last accessed 8/31/2004

4. Intel Corp., “IXP1200 Network Processor Family” December 7, 2001, http://www.intel.com/design/network/manuals/27830309.pdf, last accessed 8/31/2004

5. Abhijeet A. Joglekar, “High Capacity Network link Emulatio Using Network Processors”, May 2004, http://www.cs.utah.edu/flux/papers/joglekar-thesis-base.html, last accessed 8/31/2004

6. Ying-Dar Lin, Yi-Neng, Shun-Chin Yang, Yu-Shen Lin, “DiffServ over Network Processors: Implementation and Evaluation”, September 22, 2002, http://www.hoti.org/archive/hoti10/program/Lin_DiffServoverNP.pdf, Last accessed 8/31/2004

7. LynuxWorks, “Big-endian version of BlueCat for Intel IXP1200”May 21, 2001, http://www.electronicstalk.com/news/lyn/lyn107.html, 8/31/2004

8. “Installing IXP1200”, http://www.cse.iitb.ac.in/~sivak/ixp1200.php, last accessed 8/31/2004

9. Weidong Shi, Indrani Paul, Liang Xiao, “Implementing Real Time packet Scheduling Algorithms on IXP1200”, http://www.cc.gatech.edu/classes/AY2001/cs6235_spring/project3/presentations/ixp1200.ppt, last accessed 8/31/2004

10. Intel Corp., “IXA Education Workstation Setup Guide”, May 2002,

11. INTEL Corporation, Intel IXP1200 Network Processor Family Hardware Reference Manual, 2001

12. INTEL Corporation, Intel Internet Exchange Architecture, 2001 http://www.intel.com/design/ network/ixa.htm, last accessed not up 01/15/2004

13. Intel Corporation, Intel SDK for the IXP1200, 2001

14. Intel Corporation, Intel IXA SDK ACE Programming Framework, December 2001

15. Intel Corporation, Intel IXP1200 Network Processor Family Microcode Programmer's

16. Intel Corporation, MicroAce Design Document, 2001

17. Intel Corporation, L3 Forwarder MicroACE Design Document

18. Intel Corporation, "IXA SDK ACE Programming Framework", December 2001,

19. *http://www.cnet.com/Resources/Info/Glossary/Terms/sdram.html*, , last accessed not up October 18, 2004

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Su Wen, Code CS
   Naval Postgraduate School
   Monterey, California

4. Professor Gibson John, Code CS
   Naval Postgraduate School
   Monterey, California